Programming the Asynchronous Polycyclic Architecture

Geraldo Lino de Campos[1] and Demi Getschko[2]

[1] Escola Politécnica da Universidade de São Paulo
e-mail: glcampos@pec001.usp.ansp.br

[2] Fundação de Amparo à Pesquisa do Estado de São Paulo
e-mail: demi@fpsp.fapesp.br

Abstract The Asynchronous Polycyclic Architecture (APA) is a new processor design for numerically intensive applications. One of the main features is the ability to efficiently execute loops with recurrences and conditionals. This is achieved by the use of a special iteration register, an interactive counterpart of the stack pointer for recursive operations.

Keywords: Asynchronous Polycyclic Architecture, programming, recurrences.

1. Introduction and motivation

Development of the Asynchronous Polycyclic Architecture (APA) concept was spurred by the needs of Project Omicron, an academic research effort. The project's main goal is to design and build a supercomputer for numerical applications, with a real-world performance in the same range of the then current supercomputers. APA processors are expected to provide better sustained performance in real-world problems than standard vector processors with the same peak capacity.

Our proposed Asynchronous Polycyclic Architecture [2, 3] combines the basics of the VLIW architecture [6] with the decoupled access/execute concept [8], and with extensions for loop execution, for conditional execution of instructions, for fetching large amounts of data and for an autonomous operation of groups of functional units, and eager execution for hiding

memory latency (in *eager* execution, a instruction is executed as soon as their operands are available and there is a free unit to do the operation, even when it is not sure if the control flow will warrant the need for the instruction; in *lazy* execution an instruction is executed only when reached by the control flow).

Sections 2 describes the generic APA concept in more detail. Section 3 details the polycyclic concept of the architecture. Section 4 provides a complete programming example. Section 5 offers some concluding remarks

2. Description of the APA

The Asynchronous Polycyclic Architecture resulted from a critical analysis of the characteristics of the VLIW architecture. The detailed evolution leading to the APA can be found in [2]; it will be only summarized here.

A VLIW processor is conceptually characterized by a single thread of execution, a large number of data paths and functional units, with control planned at compile time, instructions providing enough bits to control the action of every functional unit directly and independently in each cycle, operations that require a small and predictable number of cycles to execute, and each operation can be pipelined, i. e., each functional unit can initiate a new operation in each cycle.

On examining the conceptual characterization, it becomes clear that the rationale is to have a large degree of parallelism and a simple, and therefore fast, control cycle. Closer scrutiny of the conditions above shows that they are sufficient for the goal, but most are not necessary, at least in the length stated.

First, the access/execute concept was introduced. The motivation was practical considerations on memory access. General purpose architectures rely almost invariably on caches to speed up execution, exploiting the locality of memory references. Although this is the case with general computing loads, this assumption can be wildly wrong with numerically intensive programs, since dealing with large arrays is incompatible with any realistically sized cache. This conclusion is reported for quite different machines [1, 5, 7].

The APA solves the memory access problem by decoupling the process of memory access. Two kinds of functional units are used: the first, called the *address unit*, generates and sends the required addresses to the memory subsystem; the second, called the *data reference*

unit, is responsible for reordering data words coming from memory and upon request sending them to the other functional units.

Address units may operate in two modes: single address and multiple address. In the single address mode, its role is only to receive an address calculated by an arithmetic unit and to send it to the memory subsystem; in the multiple address mode, its role is to autonomously generate the values of a set of arithmetic progressions, until a specified number of elements are generated; this mode is used for reference to (a set of) arrays. Once started, the address unit can proceed asynchronously with the main flow of control.

The second step was extending this concept of asynchronous operations to the other functional units, each with its own flow of control. Experience in programming such a machine shows that it is unduly complicated. Very few situations require this full splitting of the functional units. A hierarchical system is adequate for almost all situations: the functional units can be divided into groups, composed of a certain number of arithmetic units, each capable of forking the operation of address units.

Figure 1 shows the evolution from VLIW to APA

3. Polycyclic support

Efficient execution of loops is obtained by use of a variant of polycyclic support described in [4,7]. The programming technic will be explained for a specif instance of the architecture, with the following characteristics:

- one pipelined ALU, able to perform fixed and floating operations with a latency of 5 cycles, with forwarding registers, reducing the latency to 2 cycles for the immediate use of a result;
- two memory access queues, one each for reads and writes;
- 32 general purpose registers, called *static registers*.
- 96 *dynamic registers*. The usage of these registers are described below.

3.1– Principle of operation

The dynamic registers are divided into frames, each corresponding to the temporaries needed for each iteration of a loop. The frame size is variable, and calculated by the compiler; it must be large enough to hold all the temporaries needed during the execution of one iteration of the loop. This is an efficient renaming mechanism that allows the simultaneous

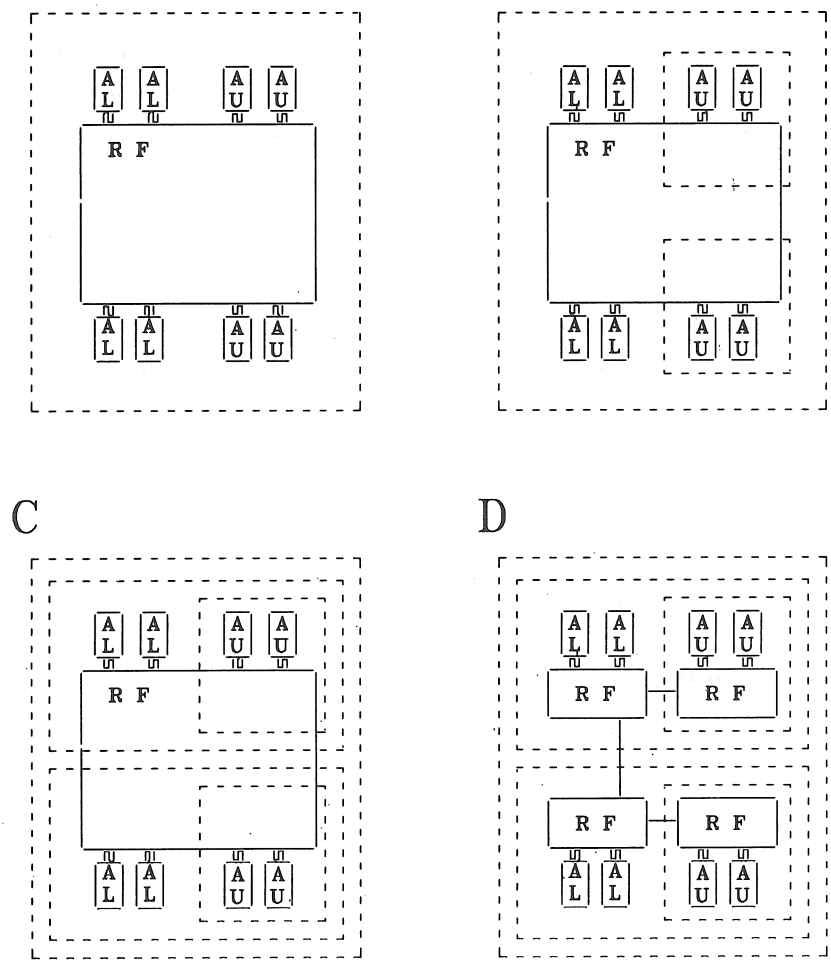

Figure 1 - Evolution from VLIW to APA architecture

In this drawing, dotted lines represent flow of control, AL stands for Arithmetic Logic Unit, and AU for Address unit.

In A, the traditional VLIW architecture. In B, evolution to the access/execute architecture. In C, the introduction of groups. Experimental analysis of programs for this architecture shows that each autonomous set may have its own copy of the register file, as shown in D. Data is exchanged via an internal bus. Other essential features for making this possible are not shown.

execution of several iterations of the same loop, hiding the ALUs latency. This is an iteractive counterpart of the stack pointer for recursive constructs.

Each frame contains the values corresponding to one iteration; old frames contains the values corresponding to older iterations. If an operation is initiated and latency would force the introduction of delays, the code can initiate a new iteration, and continue the first iteration, with values in an old frame, when executing a future iteration of the loop.

3.2– Automatic prologue and epilog

An important hardware feature is automatic prologue and epilog processing. When the execution of a loop is initiated, only the operations which refer to the current frame are executed; all the operation referring to older frames are bypassed. This phase is called prologue, and assures that only meaningful operations are done.

After the loop is executed for the expected N iterations, it is repeated M more times (this number is calculated by the compiler, and is the number of frames used by the loop). This phase is called epilogue, and the hardware operates in a complementary fashion: in the m^{th} iteration of the epilogue, only the operations that refer to frames of age m or more are executed.

This frees the compiler from generating special prologue and epilog codes, and, most important, allows it to generate code only for the general case, ignoring the special case when the loop must be executed only a few times.

4. A programming example

For the examples, the following conventions will be used:

Static registers will be represented by Rn, where n is the number of the register; dynamic register will be represented by Rn ↑ m, where n is the number of a register inside de frame, and m is the frame number, where 0 corresponds to the current frame (iteration), 1 corresponds to the values in the previous iteration, and so on.

Instructions are of the three-address format. Opcodes are self-explanatory. The source operands may be registers or the input queue, represented by Qi; the destination operand is always a register, but optionally it can be sent to the output queue, represented by Qo, as well. A flag, NEXT, indicates the end of the loop, and forces a branch to the loop's first instruction.

Before entering the loop, it is necessary to program the address units; although it is quite straightforward, it will not be shown here for space limitations. By the same reason, initialization and termination code are not shown.

As a first example, we will consider the well-known sum of products:

```
      S = 0.0
      DO 10 K = 1, N
10    S = S + A(K) * B(K)
```

The following register assignment is adopted:

R1 ↑ - A(K) * B(K);
R2 ↑ - B(K);
R5 - S

Table 1 shows the corresponding code. Version 1 is a direct compilation, without using the polycyclic features; in version 2, the NOPs following the last ADD are dropped, since the requirement of 5 cycles for reusing R5 is satisfied by the other instructions in the loop.

Version 3 uses the polycyclic feature; the ADD instruction is moved to one slot previously occupied by a NOP, and uses as operand the value of the previous iteration. As explained in the section on prologue/epilogue, the hardware does not execute this operation on the first iteration of the loop, and executes it one time more during the epilogue, so the correct number of instructions is executed.

MOV Qi, R2 ↑ 0
NOP
MUL Qi, R2 ↑ 0, R1 ↑ 0
NOP
ADD R5, R1 ↑ 0, R5
NOP
NOP
NOP
NOP
NOP, NEXT

Version 1

MOV Qi, R2 ↑ 0
NOP
MUL Qi, R2 ↑ 0, R1 ↑ 0
NOP
ADD R5, R1 ↑ 0, R5
NOP, NEXT

Version 2

MOV Qi, R2 ↑ 0
NOP
MUL Qi, R2 ↑ 0, R1 ↑ 0
ADD R5, R1 ↑ 1, R5
NOP, NEXT

Version 3

Table 1 - Code for sum of products.

In this example some NOPs are still needed to allow for the latency of 5 clock cycles; in more realistic loops, with more operations, all slots are usually filled.

To show the possibility of using loops with recurrences, lets consider the following extension to the previous loop:

```
      S = A(1) * B(1)
      DO 10 K = 2, N
      S = S + A(K) * B(K)
10    C(K) = C(K-1) + S
```

The following register assignment is adopted:

R1 ↑ - A(K) * B(K);

R2 ↑ - B(K);

R3 ↑ - S

R4 ↑ - C(K)

Table 2 shows the corresponding code. Version 1 is a direct compilation; version 2 removes the unnecessary NOPs.

Version 3 shows the polycyclic version. Another NOP slot could be used for the recurrent instruction, and each iteration still fits in five clock cycles.

5. Conclusions

The polycyclic aspect of the APA offers a simple solution for the execution of

MOV Qi, R2 ↑ 0
NOP
MUL Qi, R2 ↑ 0, R1 ↑ 0
NOP
ADD R3 ↑ 1, R1 ↑ 0, R3 ↑ 0
NOP
ADD R4 ↑ 1, R3 ↑ 0, R4 ↑ 0 [Qo]
NOP
NOP
NOP
NOP
NOP, NEXT

Version 1

MOV Qi, R2 ↑ 0
NOP
MUL Qi, R2 ↑ 0, R1 ↑ 0
NOP
ADD R3 ↑ 1, R1 ↑ 0, R3 ↑ 0
NOP
ADD R4 ↑ 1, R3 ↑ 0, R4 ↑ 0 [Qo]
NOP, NEXT

Version 2

MOV Qi, R2 ↑ 0
NOP
MUL Qi, R2 ↑ 0, R1 ↑ 0
ADD R3 ↑ 1, R1 ↑ 0, R3 ↑ 0
ADD R4 ↑ 1, R3 ↑ 0,R4 ↑ 0, NEXT

Version 3

Table 2 - Code for the second example.

loops. The increment in hardware is small, and has no impact on the cycle time.

Compared with others solutions, like software pipelining, the compiler can be much simpler. This is particularly true when considering that the automatic prologue/epilog generation allows the same code to be used even for loops of span 1, without any special consideration.

6 Bibliography

1. Abu-Sufah, W and Mahoney, A. D., "Vector Processing on the Alliant FX/8 Processor", Proc. Int'l Conf. Parallel Processing, 559-563, 1986.
2. Campos, G. L., "Asynchronous Polycyclic Architecture: an overview", Proc. of the 12th Word Computer Congress, vol I- Algorithms, Software, Architecture, Madrid, Sept 1992.
3. Campos, G. L., "Asynchronous Polycyclic Architecture", Proc. of the 6th International Conference on Parallel Processing", Lyon, Sept 1992.
4. Dehnert, J. C., Hsu, P. Y. T.., Bratt, J. P., "Overlapped Loop Support in the Cydra 5", 3rd Int. Conf. on Architectural Support for Programming Languages and Operating Systems, 26-38, April 1989
5. Diede, T., et al. "The Titan Graphics Supercomputer Architecture", IEEE Computer 21 (9):13-30, September 1988.
6. Fisher, J. A. "Very Long Instruction Word Architectures and the ELI-512", IEEE Conf. Proc. of the 10th Annual Int. Symp. on Comput. Architecture, 140-150, June 1983.
7. Rau, B. R., Yen, D. W. L. and Towle, R. A. "The Cydra 5 Departmental Supercomputer", IEEE Computer 22 (1):12-35, February 1989.
8. Smith, J. E., "Decoupled Access/Execute Architecture Computer Architectures", ACM Trans. Computer Systems, 2(4):298-308, Nov 1984.